TRABALHO PRÁTICO 2 - protocol HTTPS over TLS

Universidade de Brasília

DEPARTAMENTO DE CIÊNCIA DA COMPUTAÇÃO

DISCIPLINA DE SEGURANÇA DA COMPUTACIONAL- 2024.2

TRABALHO PRÁTICO 2 - protocol HTTPS over TLS

MEMBROS:

VINÍCIUS BOWEN - 180079239

RAMON OLIVEIRA - 242039630

Sumário

Abstract: Implementação do Servidor e Cliente HTTPS

A segurança na comunicação entre cliente e servidor é um dos principais desafios no desenvolvimento de aplicações web. O protocolo HTTPS (HyperText Transfer Protocol Secure) surge como uma solução garantindo confidencialidade, integridade e autenticidade dos dados transmitidos. Para compreender melhor seu funcionamento, este relatório apresenta a implementação de um servidor e cliente HTTPS utilizando a linguagem Python, explorando bibliotecas como `http.server`, `ssl` e `cryptography`. Além da implementação prática, são descritas estratégias de validação e testes da comunicação segura, utilizando ferramentas como OpenSSL e Wireshark.

1. Introdução: Pesquisa dos Protocolos de Segurança

1.1. Detalhamento do HTTPS, SSL e TLS

SSL (Secure Sockets Layer) e TLS (Transport Layer Security)

O SSL e seu sucessor, TLS, são protocolos essenciais para garantir a segurança das comunicações na internet. Eles utilizam criptografia para proteger dados transmitidos entre clientes e servidores, impedindo que terceiros acessem informações sensíveis como senhas e dados financeiros. O TLS surgiu como sua evolução, aprimorando a segurança e corrigindo vulnerabilidades.

Os principais objetivos desses protocolos são:
-
Confidencialidade: Utilizam criptografia simétrica e assimétrica para evitar que os dados sejam interceptados.
-
Autenticação: Certificados digitais asseguram que o servidor (e, em alguns

casos, o cliente) é legítimo.
-
Integridade dos Dados: Algoritmos de hash garantem que os dados não foram alterados durante a transmissão.

Etapas de Segurança e Principais Algoritmos

1. **Criptografia**: Usa algoritmos como AES (Advanced Encryption Standard), ChaCha20 e 3DES para criptografar os dados transmitidos.
2. **Autenticação**: Certificados digitais X.509 são utilizados para autenticar a identidade do servidor e, opcionalmente, do cliente.
3. **Troca de Chaves**: Utiliza o algoritmo Diffie-Hellman, ECDHE (Elliptic Curve Diffie-Hellman Ephemeral) ou RSA para estabelecer uma chave de sessão segura. Essa troca de chave pode sofrer um “padding” utilizando uma técnica como o OEAP.
4. **Integridade dos Dados**: Funções hash como SHA-256 garantem que os dados não foram modificados durante a transmissão.

Resumo das Versões e Evolução

- **SSL 1.0**: Nunca foi lançado publicamente devido a falhas de segurança.
- **SSL 2.0**: Lançado em 1995, mas rapidamente substituído devido a vulnerabilidades.
- **SSL 3.0**: Lançado em 1996, trouxe melhorias, mas foi considerado inseguro com o tempo.
- **TLS 1.0** (1999): Primeira versão do TLS, substituiu o SSL 3.0.
- **TLS 1.1** (2006): Melhorou a segurança contra ataques de injeção de pacotes.
- **TLS 1.2** (2008): Introduziu novos algoritmos de criptografia e hash mais seguros.
- **TLS 1.3** (2018): Tornou o handshake mais eficiente e removeu algoritmos inseguros.

1.2. HTTPS (HyperText Transfer Protocol Secure)

O HTTPS é a versão segura do HTTP, utilizando SSL ou TLS para criptografar a comunicação entre o navegador e o servidor. Esse protocolo garante que os

dados transmitidos sejam confidenciais e íntegros, protegendo contra ataques de interceptação e manipulação, como os ataques Man in the Middle. Atualmente, mais de 80% dos sites na web utilizam TLS para garantir a segurança das comunicações. O protocolo HTTPS utiliza algoritmos de criptografia robustos, como RSA, para assegurar que apenas o destinatário pretendido possa ler a mensagem, mantendo a privacidade e a integridade dos dados transmitidos.

O HTTPS evoluiu junto com os protocolos SSL e TLS, adotando as melhorias de cada versão para oferecer mais segurança e desempenho.

---

2. Implementação do Servidor e Cliente HTTPS

2.1. Visão Geral da Arquitetura Cliente/Servidor

A comunicação segura entre um cliente e um servidor HTTPS envolve a troca de informações criptografadas utilizando o protocolo TLS. O fluxo básico dessa comunicação é:

1. O cliente estabelece conexão com o servidor via HTTPS.
2. O servidor apresenta seu certificado digital (SSL/TLS).
3. O cliente verifica a autenticidade do certificado.
4. O cliente e o servidor realizam um handshake TLS para troca de chaves seguras.
5. A comunicação entre cliente e servidor ocorre de forma criptografada.

2.2. Utilização do OpenSSL e Outras Bibliotecas de Segurança

Para implementar um servidor e cliente HTTPS, utilizaremos **Python** com as bibliotecas:

- `http.server` e `ssl` para criar o servidor HTTPS.
- `requests` e `socket` para implementar o cliente HTTPS.
- `cryptography` para geração de certificados SSL/TLS via código.

- `datetime`: Usado para definir o período de validade do certificado.

2.3. Implementação Prática e Explicação do Código

A seguir, mostramos a implementação do servidor HTTPS em Python:

```
import http.server
import ssl
from cryptography import x509
from cryptography.hazmat.primitives import hashes, serialization
from cryptography.hazmat.primitives.asymmetric import rsa
import datetime

# Gera uma chave privada RSA de 2048 bits
key = rsa.generate_private_key(public_exponent=65537, key_size=2048)

# Cria um certificado autoassinado válido por 1 ano
subject = issuer = x509.Name([
    x509.NameAttribute(x509.NameOID.COUNTRY_NAME, "BR"),  # País
    x509.NameAttribute(x509.NameOID.STATE_OR_PROVINCE_NAME, "Distr
ito Federal"),  # Estado
    x509.NameAttribute(x509.NameOID.LOCALITY_NAME, "Brasília"),  # Cid
ade
    x509.NameAttribute(x509.NameOID.ORGANIZATION_NAME, "Universida
de de Brasília"),  # Organização
    x509.NameAttribute(x509.NameOID.ORGANIZATIONAL_UNIT_NAME, "ST
I"),  # Unidade Organizacional
    x509.NameAttribute(x509.NameOID.COMMON_NAME, "sti.unb.br"),  # N
ome Comum (domínio)
])

# Constrói o certificado com as informações fornecidas
cert = (
    x509.CertificateBuilder()
    .subject_name(subject)  # Define o nome do sujeito
    .issuer_name(issuer)  # Define o nome do emissor (autoassinado, então
é o mesmo que o sujeito)
    .public_key(key.public_key())  # Define a chave pública
    .serial_number(x509.random_serial_number())  # Define um número de s
```

```
érie aleatório
    .not_valid_before(datetime.datetime.utcnow())  # Define a data de início da validade
    .not_valid_after(datetime.datetime.utcnow() + datetime.timedelta(days=365))  # Define a data de término da validade (1 ano)
    .sign(key, hashes.SHA256())  # Assina o certificado com a chave privada e o algoritmo SHA-256
)

# Salva a chave privada e o certificado em arquivos
for filename, data in [("key.pem", key.private_bytes(
    encoding=serialization.Encoding.PEM,  # Codificação PEM
    format=serialization.PrivateFormat.TraditionalOpenSSL,  # Formato tradicional do OpenSSL
    encryption_algorithm=serialization.NoEncryption())),  # Sem criptografia para a chave privada
    ("cert.pem", cert.public_bytes(serialization.Encoding.PEM))]:  # Certificado em formato PEM
    with open(filename, "wb") as f:
        f.write(data)  # Escreve os dados no arquivo

# Configura o servidor HTTPS
server_address = ('localhost', 4443)  # Define o endereço do servidor (localhost) e a porta (4443)
httpd = http.server.HTTPServer(server_address, http.server.SimpleHTTPRequestHandler)  # Cria o servidor HTTP

# Configura o contexto SSL
context = ssl.SSLContext(ssl.PROTOCOL_TLS_SERVER)  # Cria um contexto SSL para o servidor
context.load_cert_chain(certfile="cert.pem", keyfile="key.pem")  # Carrega o certificado e a chave privada

# Envolve o socket do servidor com o contexto SSL
httpd.socket = context.wrap_socket(httpd.socket, server_side=True)

# Inicia o servidor HTTPS
```

```
print("Servidor HTTPS rodando em https://localhost:4443")
httpd.serve_forever()  # Mantém o servidor rodando indefinidamente
```

Instalação da Biblioteca `cryptography`

Para instalar a biblioteca `cryptography` necessária para a geração de certificados SSL/TLS, execute o seguinte comando no terminal:

```
pip install cryptography
```

Sobre a Biblioteca `cryptography`

A biblioteca `cryptography` é uma ferramenta robusta para a implementação de criptografia em Python. Ela oferece uma ampla gama de algoritmos criptográficos, incluindo RSA abordado anteriormente no seminário da disciplina, que é um dos métodos mais utilizados para a criptografia de chave pública. A `cryptography` é essencial para a geração de certificados SSL/TLS, que são fundamentais para estabelecer conexões seguras entre clientes e servidores.

Geração dos Arquivos `cert.pem` e `key.pem`

No código fornecido, a biblioteca `cryptography` é utilizada para gerar uma chave privada RSA e um certificado autoassinado. A chave privada é salva no arquivo `key.pem`, enquanto o certificado é salvo no arquivo `cert.pem`. Esses arquivos são usados pelo servidor HTTPS para criptografar a comunicação e autenticar sua identidade.

Compilação e Execução do Código

Para rodar o servidor HTTPS, execute o seguinte comando no terminal dentro do diretório onde o script Python está salvo:

```
python nome_do_arquivo.py
```

Isso iniciará o servidor e exibirá uma mensagem confirmando sua execução:

```
Servidor HTTPS rodando em https://localhost:4443
```

Agora o servidor está com conexões seguras na porta 4443.

Acessando o Servidor pelo Navegador

Captura de tela 2025-02-17 221004

1. Abra um navegador e acesse: `https://localhost:4443`
2. Como o certificado é autoassinado, o navegador exibirá um aviso de segurança.
3. Clique em **Avançado** e prossiga para o site.
4. O servidor retornará a resposta HTTP segura configurada no código.

Instalação do OpenSSL

Instale o OpenSSL no site slproweb.com e baixe a versão “Win64 OpenSSL v3.4.1 Light”:
- Com o OpenSSL instalado, abra o terminal do programa (Win64 OpenSSL Command Prompt).
- Digite o comando abaixo, no diretório onde está implementado o servidor, para verificar a criação dos certificados na etapa anterior:

```
openssl x509 -in cert.pem -text -noout
```

image

Wireshark: Analisando o Tráfego TLS

Para analisar o tráfego TLS utilizando o Wireshark, siga as instruções abaixo:

Instalação do Wireshark

1. Acesse o site oficial do Wireshark: https://www.wireshark.org/download.html.
2. Baixe a versão apropriada para o seu sistema operacional (Windows, macOS, Linux).
3. Siga as instruções de instalação fornecidas pelo instalador.

Capturando o Tráfego TLS na Porta 4443

Resumo da Análise de Tráfego HTTPS com Wireshark

1. **Identificação da Interface de Loopback**
   - Utilize o “Npcap Loopback Adapter” para capturar esse tráfego.
2. **Acessar o Servidor HTTPS**
   - Com a captura rodando, acesse no navegador: `https://localhost:4443`.
   - O Wireshark começará a registrar os pacotes HTTPS.
3. **Aplicar Filtros para Focar no Tráfego TLS**
   - Filtrar todo o tráfego TLS: `tls`
   - Filtrar pela porta 4443: `tcp.port == 4443`
   - Para visualizar o handshake, busque pacotes como “Client Hello”, “Server Hello” e “Certificate”.
   - Para acompanhar a comunicação completa, clique com o botão direito em um pacote → “Follow” → “TLS Stream”.
4. **Observando o Handshake TLS**
   - O handshake define a versão do protocolo, algoritmos de criptografia e troca de certificados.
   - No Wireshark, selecione um pacote “Client Hello” ou “Server Hello” e expanda:
     - **Transport Layer Security → Handshake Protocol**
   - Verifique a versão do TLS (ex.: TLS 1.2 ou 1.3) e as cifras negociadas.

Captura de tela 2025-02-17 215148

1. **Verificando os Dados Criptografados**
   - Após o handshake, os dados HTTP são criptografados.
   - Pacotes de “Application Data” no Wireshark contêm apenas bytes aparentemente aleatórios.
   - Não é possível visualizar requisições HTTP como GET ou POST em texto claro.

- Isso confirma que a comunicação está protegida por TLS e que terceiros não podem acessar os dados sem a chave de descriptografia.

Captura de tela 2025-02-17 220930

Análise dos Dados:
Ao clicar e expandir um desses pacotes, você perceberá que a seção de dados criptografados não apresenta os cabeçalhos HTTP (como GET, POST, etc.) e nem o corpo da mensagem em formato legível. Essa ausência de informações em texto claro indica que os dados foram criptografados conforme o esperado.

Conclusão:
O fato de os pacotes “Application Data” não exibirem conteúdo em texto plano é uma evidência de que a comunicação está protegida pelo TLS, garantindo que mesmo que alguém intercepte os pacotes, não poderá ler as informações sem a chave de descriptografia.

2.5. Preenchimento dos Campos do Certificado

Ao criar um certificado digital, diversos campos podem ser preenchidos para garantir a correta identificação da entidade proprietária do certificado. Esses campos são essenciais para assegurar autenticidade e confiança na comunicação segura. Abaixo, explicamos os principais campos e sua importância:

###

Protocolo X.509

O protocolo X.509 é um padrão para a infraestrutura de chave pública (PKI) utilizado na emissão de certificados digitais. Esses certificados são essenciais para a implementação do TLS (Transport Layer Security) na web, garantindo a segurança das comunicações entre clientes e servidores. Um certificado X.509 contém informações como a chave pública do servidor, a identidade do proprietário (nome comum, organização, país, etc.), o período de validade e a assinatura digital de uma Autoridade Certificadora (CA) confiável. Quando um navegador se conecta a um site HTTPS, ele verifica o certificado X.509 apresentado pelo servidor para assegurar que a conexão é segura e que o servidor é autêntico. O uso de certificados X.509 é fundamental para estabelecer conexões seguras e proteger os dados transmitidos na web.

- **C = Country (País)** → Indica o país onde a organização ou entidade está registrada. Exemplo: `BR` para Brasil.
- **ST = State/Province (Estado ou Província)** → Especifica o estado ou província dentro do país. Exemplo: `Distrito Federal`.
- **L = Locality (Cidade ou Localidade)** → Define a cidade onde a entidade está localizada. Exemplo: `Brasília`.
- **O = Organization (Organização)** → Nome da empresa ou entidade proprietária do certificado. Exemplo: `Universidade de Brasília`.
- **OU = Organizational Unit (Unidade Organizacional)** → Usado para definir um departamento dentro da organização. Exemplo: `TI`.
- **CN = Common Name (Nome Comum)** → Define o domínio ou nome da entidade que usará o certificado. Exemplo: `localhost` ou `sti.unb.br`.

Captura de tela 2025-02-17 221726

Como este é um trabalho da disciplina de segurança computacional, o certificado gerado não é válido para HTTPS em um ambiente real. Para que um certificado seja considerado válido, ele deve ser emitido por uma Autoridade Certificadora (CA) confiável, como a ICP-Brasil no Brasil ou a DigiCert internacionalmente. Cada navegador vem de fábrica com uma lista de autoridades certificadoras confiáveis e verifica a autenticidade do certificado através do protocolo TLS. Isso garante que o site acessado pelo usuário é certificado e que o tráfego é criptografado utilizando TLS e RSA, assegurando a confidencialidade e integridade dos dados transmitidos.

Os certificados TLS são fundamentais para a segurança na web. Eles são usados para estabelecer conexões seguras entre clientes e servidores, garantindo que os dados transmitidos sejam criptografados e protegidos contra interceptações. Um certificado TLS contém informações sobre a identidade do servidor, a chave pública usada para criptografia e a assinatura digital da CA que emitiu o certificado. Quando um navegador se conecta a um site HTTPS, ele verifica o certificado apresentado pelo servidor contra a lista de CAs confiáveis. Se o certificado for válido, a conexão segura é estabelecida, permitindo a troca de dados de forma segura e protegida.

---

3. Conclusão e Análise Final

A implementação do HTTPS melhora significativamente a segurança das aplicações web, garantindo a confidencialidade, integridade e autenticidade dos dados. Apesar dos desafios na configuração e implementação, o uso de certificados autoassinados permitiu um entendimento mais aprofundado sobre a criptografia na web. Futuramente, melhorias como a integração com autoridades certificadoras podem ser exploradas para aumentar a confiabilidade da solução.

---

4. Bibliográfia

Stallings, William. Cryptography and Network Security: Principles and Practice, Global Edition. Germany, Pearson Education, 2022.